

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [VocabularyandDefinitions.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

90 lines (68 sloc) 3.54 KB

Vocabulary and Definitions

Now that we have looked at some examples of graphs, we will more formally define a graph and its components. We already know some of these terms from our discussion of trees.

Vertex

A vertex (also called a “node”) is a fundamental part of a graph. It can have a name, which we will call the “key.” A vertex may also have additional information. We will call this additional information the “payload.”

Edge

An edge (also called an “arc”) is another fundamental part of a graph. An edge connects two vertices to show that there is a relationship between them. Edges may be one-way or two-way. If the edges in a graph are all one-way, we say that the graph is a **directed graph**, or a **digraph**. The class prerequisites graph shown above is clearly a digraph since you must take some classes before others.

Weight

Edges may be weighted to show that there is a cost to go from one vertex to another. For example in a graph of roads that connect one city to another, the weight on the edge might represent the distance between the two cities.

With those definitions in hand we can formally define a graph. A graph can be represented by G where $G=(V,E)$. For the graph G , V is a set of vertices and E is a set of edges. Each edge is a tuple (v,w) where $w,v \in V$. We can add a third component to the edge tuple to represent a weight. A subgraph s is a set of edges e and vertices v such that $e \subset E$ and $v \subset V$.

:ref:Figure 2 <fig_dgsimple>` shows another example of a simple weighted digraph. Formally we can represent this graph as the set of six vertices:

$$V = \left\{ V_0,V_1,V_2,V_3,V_4,V_5 \right\}$$

and the set of nine edges:

$$E = \left\{ \begin{array}{l} (v_0,v_1,5), (v_1,v_2,4), (v_2,v_3,9), (v_3,v_4,7), (v_4,v_0,1), \\ (v_0,v_5,2), (v_5,v_4,8), (v_3,v_5,3), (v_5,v_2,1) \end{array} \right\}$$

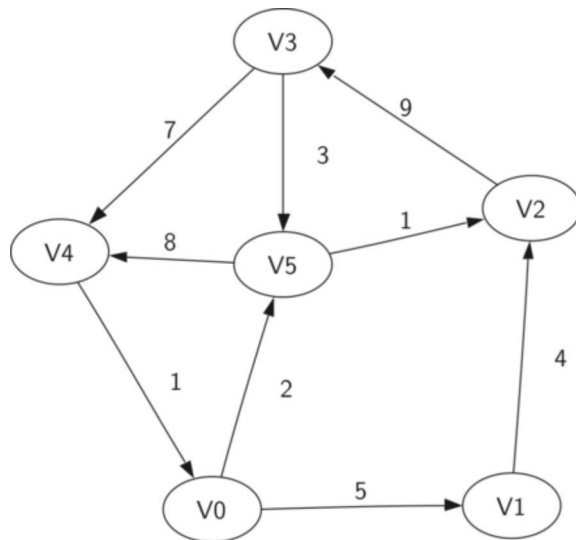


Figure 2: A Simple Example of a Directed Graph

The example graph in [:ref:`Figure 2 <fig_dgsimple>`](#) helps illustrate two other key graph terms:

Path

A path in a graph is a sequence of vertices that are connected by edges. Formally we would define a path as w_1, w_2, \dots, w_n such that $(w_i, w_{i+1}) \in E$ for all $1 \leq i \leq n-1$. The unweighted path length is the number of edges in the path, specifically $n-1$. The weighted path length is the sum of the weights of all the edges in the path. For example in [:ref:`Figure 2 <fig_dgsimple>`](#) the path from v_3 to v_1 is the sequence of vertices (v_3, v_4, v_0, v_1) . The edges are $\{(v_3, v_4, 7), (v_4, v_0, 1), (v_0, v_1, 5)\}$.

Cycle

A cycle in a directed graph is a path that starts and ends at the same vertex. For example, in [:ref:`Figure 2 <fig_dgsimple>`](#) the path (v_5, v_2, v_3, v_5) is a cycle. A graph with no cycles is called an **acyclic graph**. A directed graph with no cycles is called a **directed acyclic graph** or a **DAG**. We will see that we can solve several important problems if the problem can be represented as a DAG.

Join GitHub today

Dismiss

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾pythonds / _sources / Graphs / TheGraphAbstractDataType.rstFind fileCopy path

Fetching contributors...

Cannot retrieve contributors at this time.

36 lines (23 sloc)1.38 KB

The Graph Abstract Data Type

The graph abstract data type (ADT) is defined as follows:

- `Graph()` creates a new, empty graph.
- `addVertex(vert)` adds an instance of `Vertex` to the graph.
- `addEdge(fromVert, toVert)` Adds a new, directed edge to the graph that connects two vertices.
- `addEdge(fromVert, toVert, weight)` Adds a new, weighted, directed edge to the graph that connects two vertices.
- `getVertex(vertKey)` finds the vertex in the graph named `vertKey`.
- `getVertices()` returns the list of all vertices in the graph.
- `in` returns `True` for a statement of the form `vertex in graph`, if the given vertex is in the graph, `False` otherwise.

Beginning with the formal definition for a graph there are several ways we can implement the graph ADT in Python. We will see that there are trade-offs in using different representations to implement the ADT described above. There are two well-known implementations of a graph, the **adjacency matrix** and the **adjacency list**. We will explain both of these options, and then implement one as a Python class.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [Objectives.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

47 lines (34 sloc) 2 KB

Objectives

- To learn what a graph is and how it is used.
- To implement the **graph** abstract data type using multiple internal representations.
- To see how graphs can be used to solve a wide variety of problems

In this chapter we will study graphs. Graphs are a more general structure than the trees we studied in the last chapter; in fact you can think of a tree as a special kind of graph. Graphs can be used to represent many interesting things about our world, including systems of roads, airline flights from city to city, how the Internet is connected, or even the sequence of classes you must take to complete a major in computer science. We will see in this chapter that once we have a good representation for a problem, we can use some standard graph algorithms to solve what otherwise might seem to be a very difficult problem.

While it is relatively easy for humans to look at a road map and understand the relationships between different places, a computer has no such knowledge. However, we can also think of a road map as a graph. When we do so we can have our computer do interesting things for us. If you have ever used one of the Internet map sites, you know that a computer can find the shortest, quickest, or easiest path from one place to another.

As a student of computer science you may wonder about the courses you must take in order to get a major. A graph is good way to represent the prerequisites and other interdependencies among courses. [.ref:Figure 1 <fig1>](#) shows another graph. This one represents the courses and the order in which they must be taken to complete a major in computer science at Luther College.

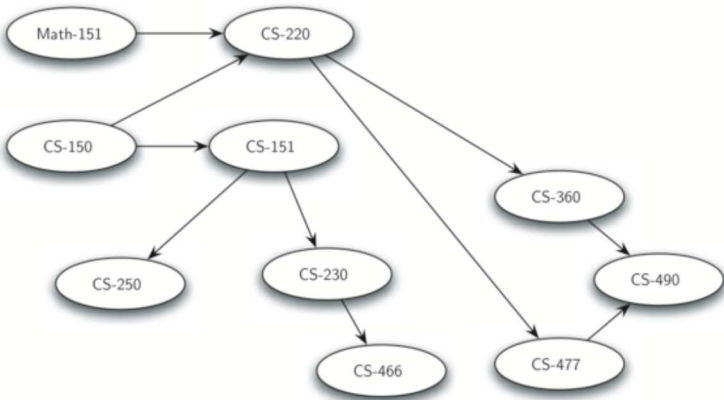


Figure 1: Prerequisites for a Computer Science Major

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [AnAdjacencyMatrix.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

42 lines (32 sloc) 2.01 KB

An Adjacency Matrix

One of the easiest ways to implement a graph is to use a two-dimensional matrix. In this matrix implementation, each of the rows and columns represent a vertex in the graph. The value that is stored in the cell at the intersection of row v and column w indicates if there is an edge from vertex v to vertex w . When two vertices are connected by an edge, we say that they are **adjacent**. [:ref:Figure 3 <fig_adjmat>](#) illustrates the adjacency matrix for the graph in [:ref:Figure 2 <fig_dgsimple>](#). A value in a cell represents the weight of the edge from vertex v to vertex w .

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

Figure 3: An Adjacency Matrix Representation for a Graph

The advantage of the adjacency matrix is that it is simple, and for small graphs it is easy to see which nodes are connected to other nodes. However, notice that most of the cells in the matrix are empty. Because most of the cells are empty we say that this matrix is “sparse.” A matrix is not a very efficient way to store sparse data. In fact, in Python you must go out of your way to even create a matrix structure like the one in [:ref:Figure 3 <fig_adjmat>](#).

The adjacency matrix is a good implementation for a graph when the number of edges is large. But what do we mean by large? How many edges would be needed to fill the matrix? Since there is one row and one column for every vertex in the graph, the number of edges required to fill the matrix is $|V|^2$. A matrix is full when every vertex is connected to every other vertex. There are few real problems that approach this sort of connectivity. The problems we will look at in this chapter all involve graphs that are sparsely connected.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

pythonds / _sources / Graphs / AnAdjacencyList.rst

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

30 lines (21 sloc) 1.21 KB

An Adjacency List

A more space-efficient way to implement a sparsely connected graph is to use an adjacency list. In an adjacency list implementation we keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to. In our implementation of the `vertex` class we will use a dictionary rather than a list where the dictionary keys are the vertices, and the values are the weights. [:ref:Figure 4 <fig_adjlist>](#) illustrates the adjacency list representation for the graph in [:ref:Figure 2 <fig_dgsimple>](#).

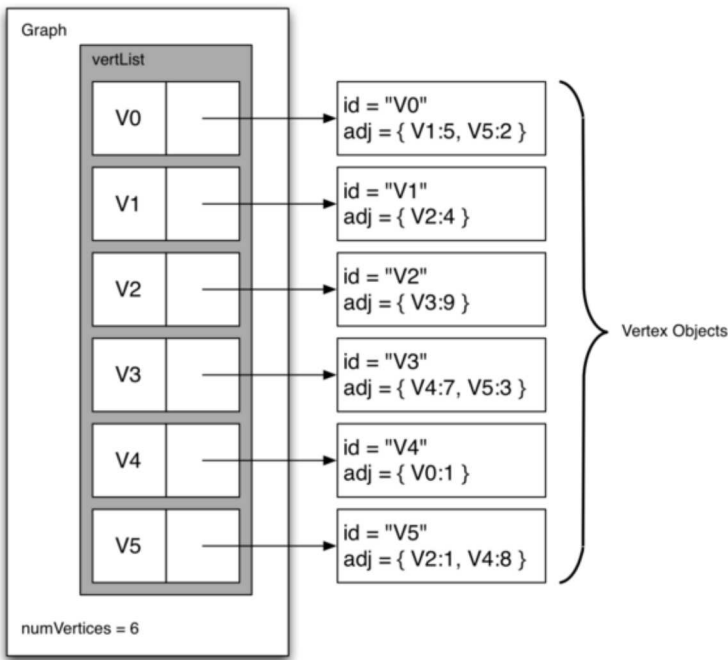


Figure 4: An Adjacency List Representation of a Graph

The advantage of the adjacency list implementation is that it allows us to compactly represent a sparse graph. The adjacency list also allows us to easily find all the links that are directly connected to a particular vertex.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [Implementation.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

143 lines (115 sloc) 4.83 KB

Implementation

Using dictionaries, it is easy to implement the adjacency list in Python. In our implementation of the Graph abstract data type we will create two classes (see [:ref:Listing 1 <lst_vertex>](#) and [:ref:Listing 2 <lst_graph>](#)), `Graph`, which holds the master list of vertices, and `Vertex`, which will represent each vertex in the graph.

Each `Vertex` uses a dictionary to keep track of the vertices to which it is connected, and the weight of each edge. This dictionary is called `connectedTo`. The listing below shows the code for the `Vertex` class. The constructor simply initializes the `id`, which will typically be a string, and the `connectedTo` dictionary. The `addNeighbor` method is used add a connection from this vertex to another. The `getConnections` method returns all of the vertices in the adjacency list, as represented by the `connectedTo` instance variable. The `getWeight` method returns the weight of the edge from this vertex to the vertex passed as a parameter.

Listing 1

```
class Vertex:
    def __init__(self, key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]
```

The `Graph` class, shown in the next listing, contains a dictionary that maps vertex names to vertex objects. In [:ref:Figure 4 <fig_adjlist>](#) this dictionary object is represented by the shaded gray box. `Graph` also provides methods for adding vertices to a graph and connecting one vertex to another. The `getVertices` method returns the names of all of the vertices in the graph. In addition, we have implemented the `__iter__` method to make it easy to iterate over all the vertex objects in a particular graph. Together, the two methods allow you to iterate over the vertices in a graph by name, or by the objects themselves.

Listing 2

```

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())


```

Using the `Graph` and `Vertex` classes just defined, the following Python session creates the graph in :ref:`Figure 2 <fig_dgsimple>`. First we create six vertices numbered 0 through 5. Then we display the vertex dictionary. Notice that for each key 0 through 5 we have created an instance of a `Vertex`. Next, we add the edges that connect the vertices together. Finally, a nested loop verifies that each edge in the graph is properly stored. You should check the output of the edge list at the end of this session against :ref:`Figure 2 <fig_dgsimple>`.

```

>>> g = Graph()
>>> for i in range(6):
...     g.addVertex(i)
>>> g.vertList
{0: <adjGraph.Vertex instance at 0x41e18>,
 1: <adjGraph.Vertex instance at 0x7f2b0>,
 2: <adjGraph.Vertex instance at 0x7f288>,
 3: <adjGraph.Vertex instance at 0x7f350>,
 4: <adjGraph.Vertex instance at 0x7f328>,
 5: <adjGraph.Vertex instance at 0x7f300>}
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> for v in g:
...     for w in v.getConnections():
...         print("( %s , %s )" % (v.getId(), w.getId()))
...
( 0 , 5 )
( 0 , 1 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )

```


 Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Join GitHub today

Dismiss

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾[pythonds](#) / [_sources](#) / [Graphs](#) / [TheWordLadderProblem.rst](#)

Find fileCopy path

Fetching contributors...

Cannot retrieve contributors at this time.

42 lines (31 sloc)1.56 KB

The Word Ladder Problem

To begin our study of graph algorithms let’s consider the following puzzle called a word ladder. Transform the word “FOOL” into the word “SAGE”. In a word ladder puzzle you must make the change occur gradually by changing one letter at a time. At each step you must transform one word into another word, you are not allowed to transform a word into a non-word. The word ladder puzzle was invented in 1878 by Lewis Carroll, the author of *Alice in Wonderland*. The following sequence of words shows one possible solution to the problem posed above.

FOOL
POOL
POLL
POLE
PALE
SALE
SAGE

There are many variations of the word ladder puzzle. For example you might be given a particular number of steps in which to accomplish the transformation, or you might need to use a particular word. In this section we are interested in figuring out the smallest number of transformations needed to turn the starting word into the ending word.

Not surprisingly, since this chapter is on graphs, we can solve this problem using a graph algorithm. Here is an outline of where we are going:

- Represent the relationships between the words as a graph.
- Use the graph algorithm known as breadth first search to find an efficient path from the starting word to the ending word.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master

pythonds / _sources / Graphs / BuildingtheWordLadderGraph.rst

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

102 lines (81 sloc) 4.3 KB

Building the Word Ladder Graph

Our first problem is to figure out how to turn a large collection of words into a graph. What we would like is to have an edge from one word to another if the two words are only different by a single letter. If we can create such a graph, then any path from one word to another is a solution to the word ladder puzzle. [.ref:Figure 1 <fig_wordladder>](#) shows a small graph of some words that solve the FOOL to SAGE word ladder problem. Notice that the graph is an undirected graph and that the edges are unweighted.

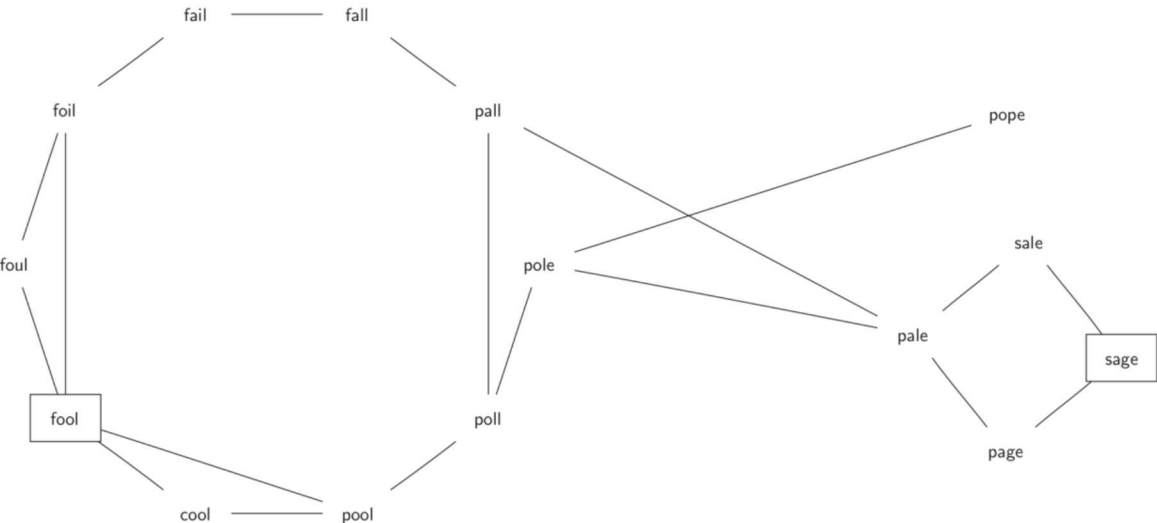


Figure 1: A Small Word Ladder Graph

We could use several different approaches to create the graph we need to solve this problem. Let's start with the assumption that we have a list of words that are all the same length. As a starting point, we can create a vertex in the graph for every word in the list. To figure out how to connect the words, we could compare each word in the list with every other. When we compare we are looking to see how many letters are different. If the two words in question are different by only one letter, we can create an edge between them in the graph. For a small set of words that approach would work fine; however let's suppose we have a list of 5,110 words. Roughly speaking, comparing one word to every other word on the list is an $O(n^2)$ algorithm. For 5,110 words, n^2 is more than 26 million comparisons.

We can do much better by using the following approach. Suppose that we have a huge number of buckets, each of them with a four-letter word on the outside, except that one of the letters in the label has been replaced by an underscore. For example, consider [.ref:Figure 2 <fig_wordbucket>](#), we might have a bucket labeled "pop_." As we process each word in our list we compare the word with each bucket, using the '_' as a wildcard, so both "pope" and "pops" would match "pop_." Every time we find a matching bucket, we put our word in that bucket. Once we have all the words in the appropriate buckets we know that all the words in the bucket must be connected.

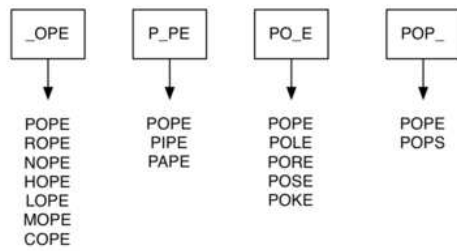


Figure 2: Word Buckets for Words That are Different by One Letter

In Python, we can implement the scheme we have just described by using a dictionary. The labels on the buckets we have just described are the keys in our dictionary. The value stored for that key is a list of words. Once we have the dictionary built we can create the graph. We start our graph by creating a vertex for each word in the graph. Then we create edges between all the vertices we find for words found under the same key in the dictionary. [.ref:Listing 1 <lst_wordbucket1>](#) shows the Python code required to build the graph.

Listing 1

```

from pythonds.graphs import Graph

def buildGraph(wordFile):
    d = {}
    g = Graph()
    wfile = open(wordFile, 'r')
    # create buckets of words that differ by one letter
    for line in wfile:
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    # add vertices and edges for words in the same bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1, word2)
    return g

```

Since this is our first real-world graph problem, you might be wondering how sparse is the graph? The list of four-letter words we have for this problem is 5,110 words long. If we were to use an adjacency matrix, the matrix would have $5,110 * 5,110 = 26,112,100$ cells. The graph constructed by the `buildGraph` function has exactly 53,286 edges, so the matrix would have only 0.20% of the cells filled! That is a very sparse matrix indeed.



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: [master](#) ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [ImplementingBreadthFirstSearch.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

175 lines (129 sloc) 6.97 KB

Implementing Breadth First Search

With the graph constructed we can now turn our attention to the algorithm we will use to find the shortest solution to the word ladder problem. The graph algorithm we are going to use is called the “breadth first search” algorithm. **Breadth first search (BFS)** is one of the easiest algorithms for searching a graph. It also serves as a prototype for several other important graph algorithms that we will study later.

Given a graph G and a starting vertex s , a breadth first search proceeds by exploring edges in the graph to find all the vertices in G for which there is a path from s . The remarkable thing about a breadth first search is that it finds *all* the vertices that are a distance k from s before it finds *any* vertices that are a distance $k+1$. One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time. A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.

To keep track of its progress, BFS colors each of the vertices white, gray, or black. All the vertices are initialized to white when they are constructed. A white vertex is an undiscovered vertex. When a vertex is initially discovered it is colored gray, and when BFS has completely explored a vertex it is colored black. This means that once a vertex is colored black, it has no white vertices adjacent to it. A gray node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore.

The breadth first search algorithm shown in [:ref:`Listing 2 <lst_wordbucket2>`](#) below uses the adjacency list graph representation we developed earlier. In addition it uses a `queue`, a crucial point as we will see, to decide which vertex to explore next.

In addition the BFS algorithm uses an extended version of the `vertex` class. This new vertex class adds three new instance variables: `distance`, `predecessor`, and `color`. Each of these instance variables also has the appropriate getter and setter methods. The code for this expanded `Vertex` class is included in the `pythonds` package, but we will not show it to you here as there is nothing new to learn by seeing the additional instance variables.

BFS begins at the starting vertex s and colors `start` gray to show that it is currently being explored. Two other values, the distance and the predecessor, are initialized to 0 and `None` respectively for the starting vertex. Finally, `start` is placed on a `queue`. The next step is to begin to systematically explore vertices at the front of the queue. We explore each new node at the front of the queue by iterating over its adjacency list. As each node on the adjacency list is examined its color is checked. If it is white, the vertex is unexplored, and four things happen:

1. The new, unexplored vertex `nbr`, is colored gray.
2. The predecessor of `nbr` is set to the current node `currentVert`
3. The distance to `nbr` is set to the distance to `currentVert` + 1
4. `nbr` is added to the end of a queue. Adding `nbr` to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of `currentVert` have been explored.

Listing 2

```

from pythonds.graphs import Graph, Vertex
from pythonds.basic import Queue

def bfs(g, start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = Queue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                nbr.setDistance(currentVert.getDistance() + 1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
        currentVert.setColor('black')

```

Let's look at how the `bfs` function would construct the breadth first tree corresponding to the graph in [:ref:Figure 1 <fig_wordladder>](#). Starting from fool we take all nodes that are adjacent to fool and add them to the tree. The adjacent nodes include pool, foil, foul, and cool. Each of these nodes are added to the queue of new nodes to expand. [:ref:Figure 3 <fig_bfs1>](#) shows the state of the in-progress tree along with the queue after this step.

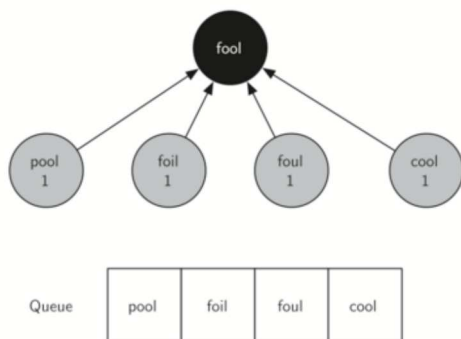


Figure 3: The First Step in the Breadth First Search

In the next step `bfs` removes the next node (`pool`) from the front of the queue and repeats the process for all of its adjacent nodes. However, when `bfs` examines the node `cool`, it finds that the color of `cool` has already been changed to gray. This indicates that there is a shorter path to `cool` and that `cool` is already on the queue for further expansion. The only new node added to the queue while examining `pool` is `poll`. The new state of the tree and queue is shown in [:ref:Figure 4 <fig_bfs2>](#).

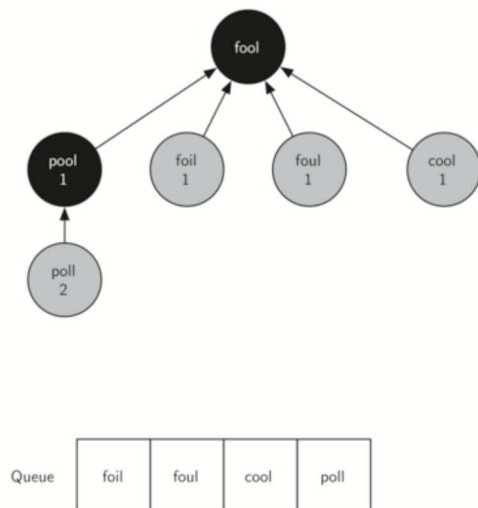


Figure 4: The Second Step in the Breadth First Search

The next vertex on the queue is `foil`. The only new node that `foil` can add to the tree is `fail`. As `bfs` continues to process the queue, neither of the next two nodes add anything new to the queue or the tree. [:ref:Figure 5 <fig_bfs3>](#) shows the tree and the queue after expanding all the vertices on the second level of the tree.

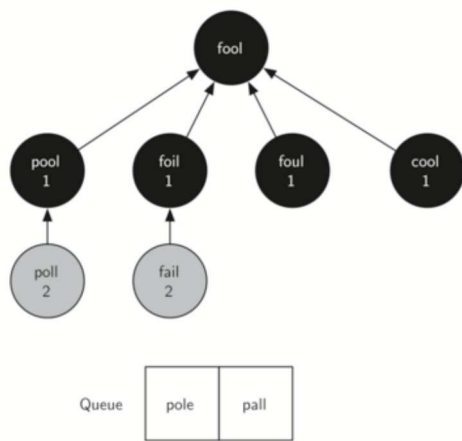


Figure 5: Breadth First Search Tree After Completing One Level

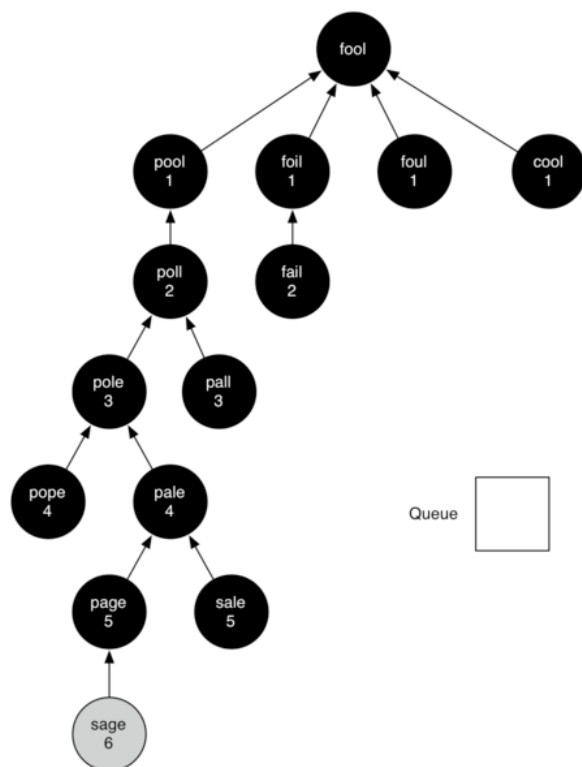


Figure 6: Final Breadth First Search Tree


You should continue to work through the algorithm on your own so that you are comfortable with how it works. [:ref:`Figure 6 <fig_bfsDone>`](#) shows the final breadth first search tree after all the vertices in [:ref:`Figure 3 <fig_wordladder>`](#) have been expanded. The amazing thing about the breadth first search solution is that we have not only solved the FOOL-SAGE problem we started out with, but we have solved many other problems along the way. We can start at any vertex in the breadth first search tree and follow the predecessor arrows back to the root to find the shortest word ladder from any word back to fool. The function below ([:ref:`Listing 3 <lst_wordbucket3>`](#)) shows how to follow the predecessor links to print out the word ladder.

Listing 3

```

def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())

traverse(g.getVertex('sage'))
  
```

 Please note that GitHub no longer supports old versions of Firefox. We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Join GitHub today

Dismiss

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾[pythonds](#) / [_sources](#) / [Graphs](#) / [BreadthFirstSearchAnalysis.rst](#)

Find fileCopy path

Fetching contributors...

Cannot retrieve contributors at this time.

30 lines (24 sloc) 1.62 KB

Breadth First Search Analysis

Before we continue with other graph algorithms let us analyze the run time performance of the breadth first search algorithm. The first thing to observe is that the while loop is executed, at most, one time for each vertex in the graph $|V|$. You can see that this is true because a vertex must be white before it can be examined and added to the queue. This gives us $O(V)$ for the while loop. The for loop, which is nested inside the while is executed at most once for each edge in the graph, $|E|$. The reason is that every vertex is dequeued at most once and we examine an edge from node u to node v only when node u is dequeued. This gives us $O(E)$ for the for loop. combining the two loops gives us $O(V + E)$.

Of course doing the breadth first search is only part of the task. Following the links from the starting node to the goal node is the other part of the task. The worst case for this would be if the graph was a single long chain. In this case traversing through all of the vertices would be $O(V)$. The normal case is going to be some fraction of $|V|$ but we would still write $O(V)$.

Finally, at least for this problem, there is the time required to build the initial graph. We leave the analysis of the `buildGraph` function as an exercise for you.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾pythonds / _sources / Graphs / TheKnightsTourProblem.rst

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

32 lines (24 sloc)1.49 KB

The Knight’s Tour Problem

Another classic problem that we can use to illustrate a second common graph algorithm is called the “knight’s tour.” The knight’s tour puzzle is played on a chess board with a single chess piece, the knight. The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once. One such sequence is called a “tour.” The knight’s tour puzzle has fascinated chess players, mathematicians and computer scientists alike for many years. The upper bound on the number of possible legal tours for an eight-by-eight chessboard is known to be 1.305×10^{35} ; however, there are even more possible dead ends. Clearly this is a problem that requires some real brains, some real computing power, or both.

Although researchers have studied many different algorithms to solve the knight’s tour problem, a graph search is one of the easiest to understand and program. Once again we will solve the problem using two main steps:

- Represent the legal moves of a knight on a chessboard as a graph.
- Use a graph algorithm to find a path of length $\text{rows} \times \text{columns} - 1$ where every vertex on the graph is visited exactly once.

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Dismiss

Branch: master ▾[pythonds](#) / [_sources](#) / [Graphs](#) / [BuildingtheKnightsTourGraph.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

101 lines (72 sloc) 3.49 KB

Building the Knight’s Tour Graph

To represent the knight’s tour problem as a graph we will use the following two ideas: Each square on the chessboard can be represented as a node in the graph. Each legal move by the knight can be represented as an edge in the graph. [:ref:Figure 1 <fig_knightmoves>](#) illustrates the legal moves by a knight and the corresponding edges in a graph.

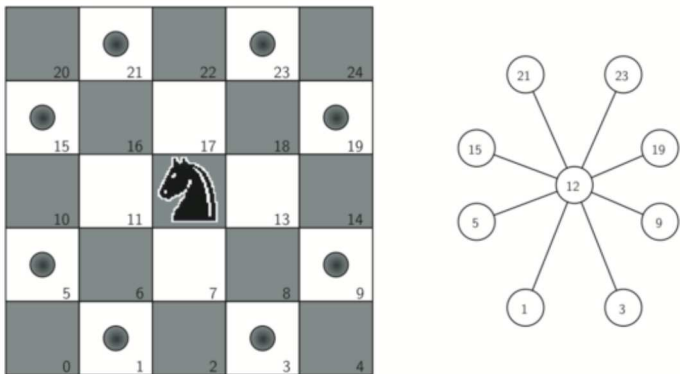


Figure 1: Legal Moves for a Knight on Square 12, and the Corresponding Graph

To build the full graph for an n-by-n board we can use the Python function shown in [:ref:Listing 1 <lst_knighttour1>](#). The `knightGraph` function makes one pass over the entire board. At each square on the board the `knightGraph` function calls a helper, `genLegalMoves`, to create a list of legal moves for that position on the board. All legal moves are then converted into edges in the graph. Another helper function `posToNodeId` converts a location on the board in terms of a row and a column into a linear vertex number similar to the vertex numbers shown in [:ref:Figure 1 <fig_knightmoves>](#).

Listing 1

```
from pythonds.graphs import Graph

def knightGraph.bdSize):
    ktGraph = Graph()
    for row in range.bdSize):
        for col in range.bdSize):
            nodeId = posToNodeId(row,col,bdSize)
            newPositions = genLegalMoves(row,col,bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0],e[1],bdSize)
                ktGraph.addEdge(nodeId,nid)
    return ktGraph

def posToNodeId(row, column, board_size):
```

```
return (row * board_size) + column
```

The `genLegalMoves` function (:ref:`Listing 2 <lst_knighttour2>`) takes the position of the knight on the board and generates each of the eight possible moves. The `legalCoord` helper function (:ref:`Listing 2 <lst_knighttour2>`) makes sure that a particular move that is generated is still on the board.

Listing 2

```
def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                   ( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and \
            legalCoord(newY,bdSize):
            newMoves.append((newX,newY))
    return newMoves

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```

:ref:`Figure 2 <fig_bigknight>` shows the complete graph of possible moves on an eight-by-eight board. There are exactly 336 edges in the graph. Notice that the vertices corresponding to the edges of the board have fewer connections (legal moves) than the vertices in the middle of the board. Once again we can see how sparse the graph is. If the graph was fully connected there would be 4,096 edges. Since there are only 336 edges, the adjacency matrix would be only 8.2 percent full.

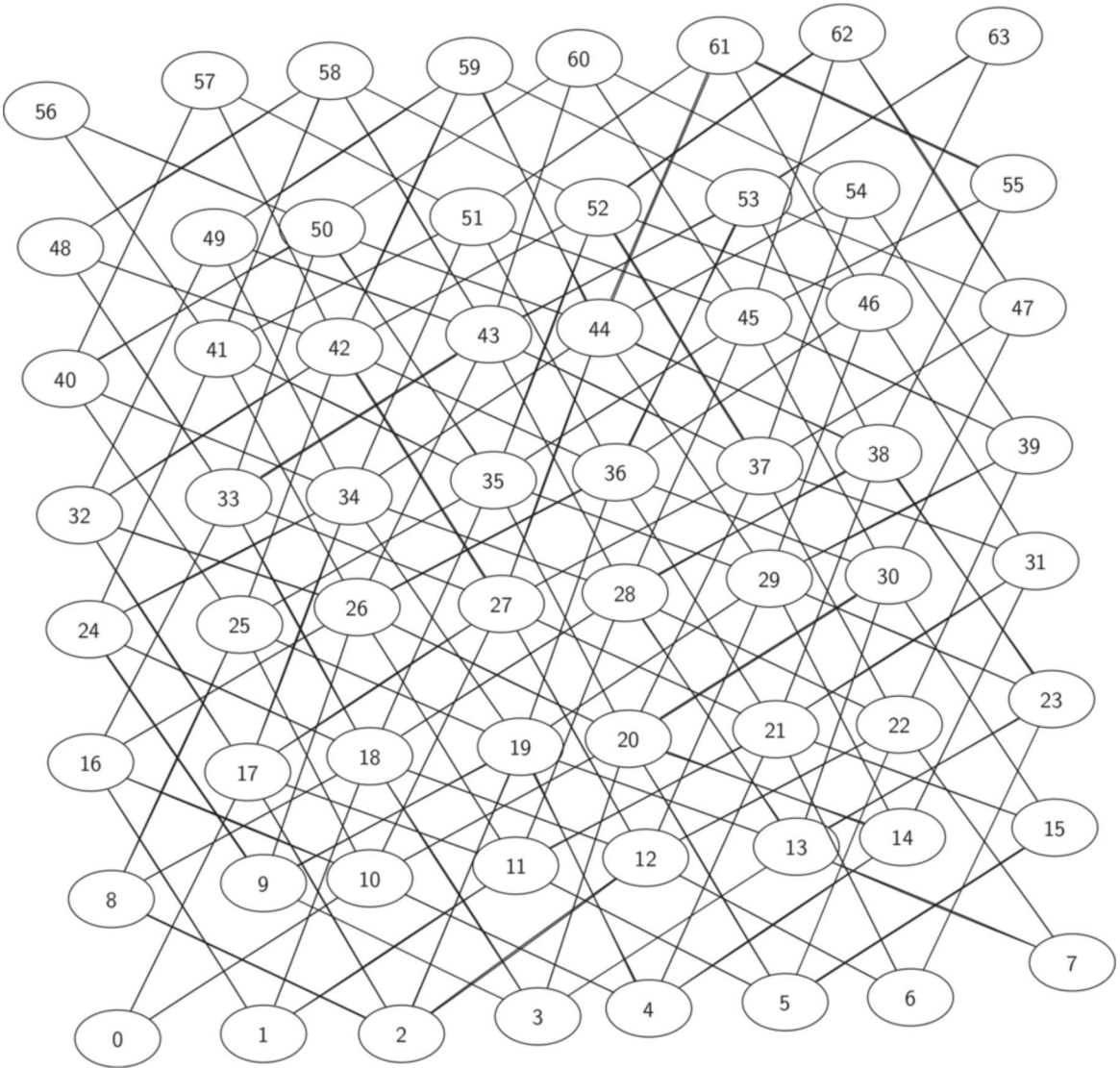


Figure 2: All Legal Moves for a Knight on an 8 \times 8 Chessboard



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: [master](#) ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [ImplementingKnightsTour.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

184 lines (125 sloc) 6.33 KB

Implementing Knight's Tour

The search algorithm we will use to solve the knight's tour problem is called **depth first search (DFS)**. Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible. In this section we will look at two algorithms that implement a depth first search. The first algorithm we will look at directly solves the knight's tour problem by explicitly forbidding a node to be visited more than once. The second implementation is more general, but allows nodes to be visited more than once as the tree is constructed. The second version is used in subsequent sections to develop additional graph algorithms.

The depth first exploration of the graph is exactly what we need in order to find a path that has exactly 63 edges. We will see that when the depth first search algorithm finds a dead end (a place in the graph where there are no more moves possible) it backs up the tree to the next deepest vertex that allows it to make a legal move.

The `knightTour` function takes four parameters: `n`, the current depth in the search tree; `path`, a list of vertices visited up to this point; `u`, the vertex in the graph we wish to explore; and `limit` the number of nodes in the path. The `knightTour` function is recursive. When the `knightTour` function is called, it first checks the base case condition. If we have a path that contains 64 vertices, we return from `knightTour` with a status of `True`, indicating that we have found a successful tour. If the path is not long enough we continue to explore one level deeper by choosing a new vertex to explore and calling `knightTour` recursively for that vertex.

DFS also uses colors to keep track of which vertices in the graph have been visited. Unvisited vertices are colored white, and visited vertices are colored gray. If all neighbors of a particular vertex have been explored and we have not yet reached our goal length of 64 vertices, we have reached a dead end. When we reach a dead end we must backtrack. Backtracking happens when we return from `knightTour` with a status of `False`. In the breadth first search we used a queue to keep track of which vertex to visit next. Since depth first search is recursive, we are implicitly using a stack to help us with our backtracking. When we return from a call to `knightTour` with a status of `False`, in line 11, we remain inside the `while` loop and look at the next vertex in `nbrList`.

Listing 3

```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
    u.setColor('gray')
    path.append(u)
    if n < limit:
        nbrList = list(u.getConnections())
        i = 0
        done = False
        while i < len(nbrList) and not done:
            if nbrList[i].getColor() == 'white':
                done = knightTour(n+1, path, nbrList[i], limit)
```

```

        i = i + 1
    if not done: # prepare to backtrack
        path.pop()
        u.setColor('white')
    else:
        done = True
    return done

```

Let's look at a simple example of `knightTour` in action. You can refer to the figures below to follow the steps of the search. For this example we will assume that the call to the `getConnections` method on line 6 orders the nodes in alphabetical order. We begin by calling `knightTour(0, path, A, 6)`

`knightTour` starts with node A [:ref:'Figure 3 <fig_kta>'](#). The nodes adjacent to A are B and D. Since B is before D alphabetically, DFS selects B to expand next as shown in [:ref:'Figure 4 <fig_ktb>'](#). Exploring B happens when `knightTour` is called recursively. B is adjacent to C and D, so `knightTour` elects to explore C next. However, as you can see in [:ref:'Figure 5 <fig_ktc>'](#) node C is a dead end with no adjacent white nodes. At this point we change the color of node C back to white. The call to `knightTour` returns a value of `False`. The return from the recursive call effectively backtracks the search to vertex B (see [:ref:'Figure 6 <fig_ktd>'](#)). The next vertex on the list to explore is vertex D, so `knightTour` makes a recursive call moving to node D (see [:ref:'Figure 7 <fig_kte>'](#)). From vertex D on, `knightTour` can continue to make recursive calls until we get to node C again (see [:ref:'Figure 8 <fig_ktf>'](#), [:ref:'Figure 9 <fig_ktg>'](#), and [:ref:'Figure 10 <fig_kth>'](#)). However, this time when we get to node C the test `n < limit` fails so we know that we have exhausted all the nodes in the graph. At this point we can return `True` to indicate that we have made a successful tour of the graph. When we return the list, `path` has the values `[A,B,D,E,F,C]`, which is the the order we need to traverse the graph to visit each node exactly once.

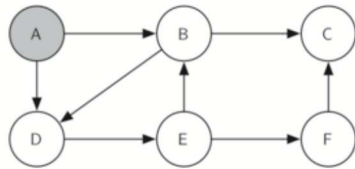


Figure 3: Start with node A

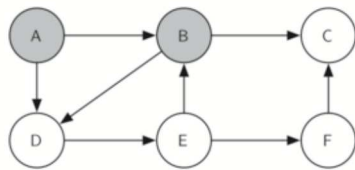


Figure 4: Explore B

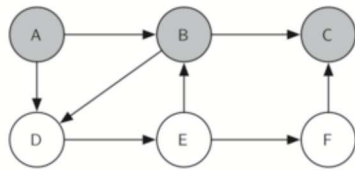


Figure 5: Node C is a dead end

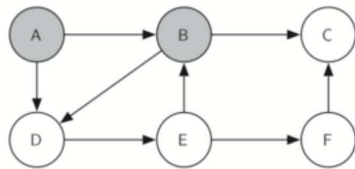


Figure 6: Backtrack to B

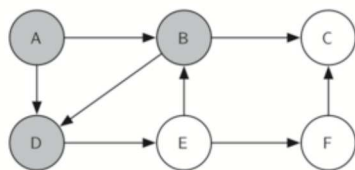


Figure 7: Explore D

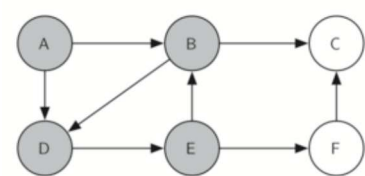


Figure 8: Explore E

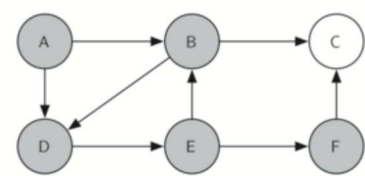


Figure 9: Explore F

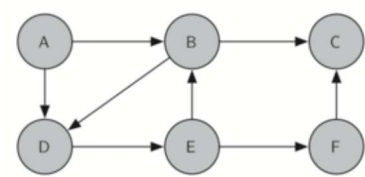


Figure 10: Finish

[:ref:Figure 11 <fig_tour>](#) shows you what a complete tour around an eight-by-eight board looks like. There are many possible tours; some are symmetric. With some modification you can make circular tours that start and end at the same square.

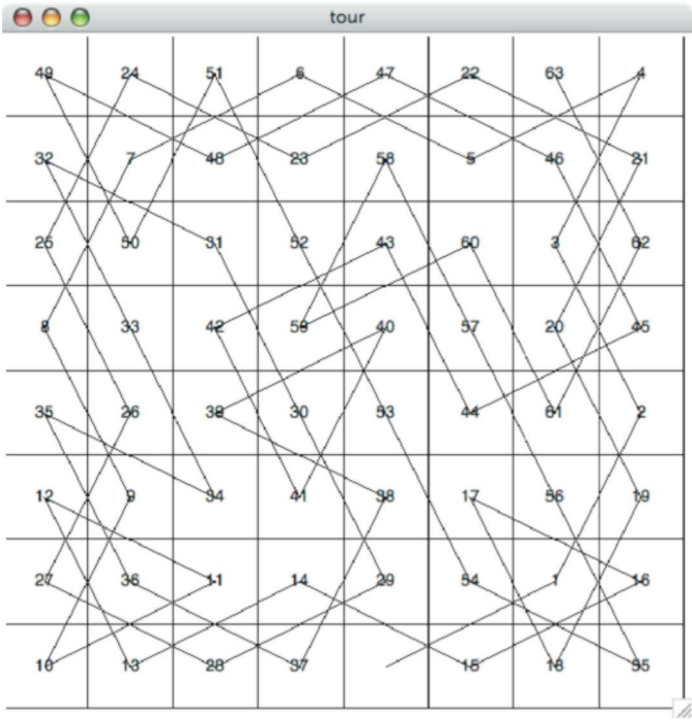
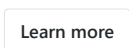


Figure 11: A Complete Tour of the Board



2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Figure 13: Number of Possible Moves for Each Square

We have already seen that the number of nodes in a binary tree of height N is $2^{N+1}-1$. For a tree with nodes that may have up to eight children instead of two the number of nodes is much larger. Because the branching factor of each node is variable, we could estimate the number of nodes using an average branching factor. The important thing to note is that this algorithm is exponential: $k^{N+1}-1$, where k is the average branching factor for the board. Let's look at how rapidly this grows! For a board that is 5×5 the tree will be 25 levels deep, or $N = 24$ counting the first level as level 0. The average branching factor is $k = 3.8$. So the number of nodes in the search tree is $3.8^{25}-1$ or 3.12×10^{14} . For a 6×6 board, $k = 4.4$, there are 1.5×10^{23} nodes, and for a regular 8×8 chess board, $k = 5.25$, there are 1.3×10^{46} . Of course, since there are multiple solutions to the problem we won't have to explore every single node, but the fractional part of the nodes we do have to explore is just a constant multiplier which does not change the exponential nature of the problem. We will leave it as an exercise for you to see if you can express k as a function of the board size.

Luckily there is a way to speed up the eight-by-eight case so that it runs in under one second. In the listing below we show the code that speeds up the `knightTour`. This function (see [:ref:Listing 4 <lst_avail>](#)), called `orderByAvail` will be used in place of the call to `u.getConnections` in the code previously shown above. The critical line in the `orderByAvail` function is line 10. This line ensures that we select the vertex to go next that has the fewest available moves. You might think this is really counter productive; why not select the node that has the most available moves? You can try that approach easily by running the program yourself and inserting the line `resList.reverse()` right after the sort.

The problem with using the vertex with the most available moves as your next vertex on the path is that it tends to have the knight visit the middle squares early on in the tour. When this happens it is easy for the knight to get stranded on one side of the board where it cannot reach unvisited squares on the other side of the board. On the other hand, visiting the squares with the fewest available moves first pushes the knight to visit the squares around the edges of the board first. This ensures that the knight will visit the hard-to-reach corners early and can use the middle squares to hop across the board only when necessary. Utilizing this kind of knowledge to speed up an algorithm is called a heuristic. Humans use heuristics every day to help make decisions, heuristic searches are often used in the field of artificial intelligence. This particular heuristic is called Warnsdorff's algorithm, named after H. C. Warnsdorff who published his idea in 1823.

Listing 4

```
def orderByAvail(n):
    resList = []
    for v in n.getConnections():
        if v.getColor() == 'white':
            c = 0
            for w in v.getConnections():
                if w.getColor() == 'white':
                    c = c + 1
            resList.append((c,v))
    resList.sort(key=lambda x: x[0])
    return [y[1] for y in resList]
```



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: [master](#) ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [GeneralDepthFirstSearch.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

241 lines (175 sloc) 9.04 KB

General Depth First Search

The knight's tour is a special case of a depth first search where the goal is to create the deepest depth first tree, without any branches. The more general depth first search is actually easier. Its goal is to search as deeply as possible, connecting as many nodes in the graph as possible and branching where necessary.

It is even possible that a depth first search will create more than one tree. When the depth first search algorithm creates a group of trees we call this a **depth first forest**. As with the breadth first search our depth first search makes use of predecessor links to construct the tree. In addition, the depth first search will make use of two additional instance variables in the `Vertex` class. The new instance variables are the discovery and finish times. The discovery time tracks the number of steps in the algorithm before a vertex is first encountered. The finish time is the number of steps in the algorithm before a vertex is colored black. As we will see after looking at the algorithm, the discovery and finish times of the nodes provide some interesting properties we can use in later algorithms.

The code for our depth first search is shown in [:ref:'Listing 5 <lst_dfsgeneral>'](#). Since the two functions `dfs` and its helper `dfsvisit` use a variable to keep track of the time across calls to `dfsvisit` we chose to implement the code as methods of a class that inherits from the `Graph` class. This implementation extends the graph class by adding a `time` instance variable and the two methods `dfs` and `dfsvisit`. Looking at line 11 you will notice that the `dfs` method iterates over all of the vertices in the graph calling `dfsvisit` on the nodes that are white. The reason we iterate over all the nodes, rather than simply searching from a chosen starting node, is to make sure that all nodes in the graph are considered and that no vertices are left out of the depth first forest. It may look unusual to see the statement `for aVertex in self`, but remember that in this case `self` is an instance of the `DFSGraph` class, and iterating over all the vertices in an instance of a graph is a natural thing to do.

Listing 5

```
from pythonds.graphs import Graph
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
```

```

for nextVertex in startVertex.getConnections():
    if nextVertex.getColor() == 'white':
        nextVertex.setPred(startVertex)
        self.dfsvisit(nextVertex)
startVertex.setColor('black')
self.time += 1
startVertex.setFinish(self.time)

```

Although our implementation of `bfs` was only interested in considering nodes for which there was a path leading back to the start, it is possible to create a breadth first forest that represents the shortest path between all pairs of nodes in the graph. We leave this as an exercise. In our next two algorithms we will see why keeping track of the depth first forest is important.

The `dfsvisit` method starts with a single vertex called `startVertex` and explores all of the neighboring white vertices as deeply as possible. If you look carefully at the code for `dfsvisit` and compare it to breadth first search, what you should notice is that the `dfsvisit` algorithm is almost identical to `bfs` except that on the last line of the inner `for` loop, `dfsvisit` calls itself recursively to continue the search at a deeper level, whereas `bfs` adds the node to a queue for later exploration. It is interesting to note that where `bfs` uses a queue, `dfsvisit` uses a stack. You don't see a stack in the code, but it is implicit in the recursive call to `dfsvisit`.

The following sequence of figures illustrates the depth first search algorithm in action for a small graph. In these figures, the dotted lines indicate edges that are checked, but the node at the other end of the edge has already been added to the depth first tree. In the code this test is done by checking that the color of the other node is non-white.

The search begins at vertex A of the graph (:ref:`Figure 14 <fig_gdfsa>`). Since all of the vertices are white at the beginning of the search the algorithm visits vertex A. The first step in visiting a vertex is to set the color to gray, which indicates that the vertex is being explored and the discovery time is set to 1. Since vertex A has two adjacent vertices (B, D) each of those need to be visited as well. We'll make the arbitrary decision that we will visit the adjacent vertices in alphabetical order.

Vertex B is visited next (:ref:`Figure 15 <fig_gdfsb>`), so its color is set to gray and its discovery time is set to 2. Vertex B is also adjacent to two other nodes (C, D) so we will follow the alphabetical order and visit node C next.

Visiting vertex C (:ref:`Figure 16 <fig_gdfsc>`) brings us to the end of one branch of the tree. After coloring the node gray and setting its discovery time to 3, the algorithm also determines that there are no adjacent vertices to C. This means that we are done exploring node C and so we can color the vertex black, and set the finish time to 4. You can see the state of our search at this point in :ref:`Figure 17 <fig_gdfsd>`.

Since vertex C was the end of one branch we now return to vertex B and continue exploring the nodes adjacent to B. The only additional vertex to explore from B is D, so we can now visit D (:ref:`Figure 18 <fig_gdfse>`) and continue our search from vertex D. Vertex D quickly leads us to vertex E (:ref:`Figure 19 <fig_gdfsf>`). Vertex E has two adjacent vertices, B and F. Normally we would explore these adjacent vertices alphabetically, but since B is already colored gray the algorithm recognizes that it should not visit B since doing so would put the algorithm in a loop! So exploration continues with the next vertex in the list, namely F (:ref:`Figure 20 <fig_gdfsg>`).

Vertex F has only one adjacent vertex, C, but since C is colored black there is nothing else to explore, and the algorithm has reached the end of another branch. From here on, you will see in :ref:`Figure 21 <fig_gdfsh>` through :ref:`Figure 25 <fig_gdfsl>` that the algorithm works its way back to the first node, setting finish times and coloring vertices black.

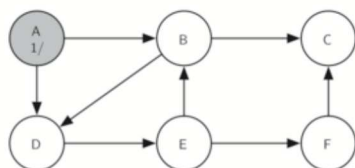


Figure 14: Constructing the Depth First Search Tree-10

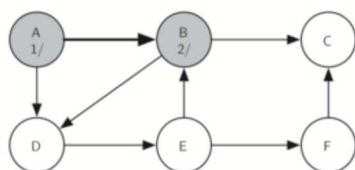


Figure 15: Constructing the Depth First Search Tree-11

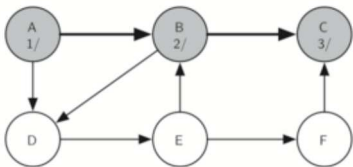


Figure 16: Constructing the Depth First Search Tree-12

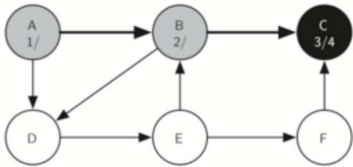


Figure 17: Constructing the Depth First Search Tree-13

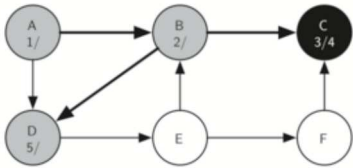


Figure 18: Constructing the Depth First Search Tree-14

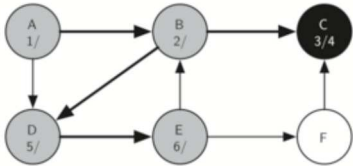


Figure 19: Constructing the Depth First Search Tree-15

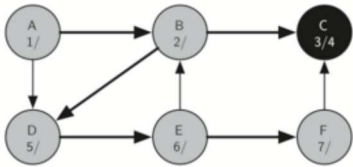


Figure 20: Constructing the Depth First Search Tree-16

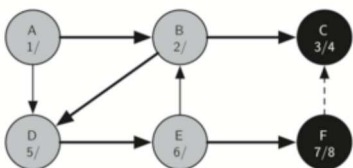


Figure 21: Constructing the Depth First Search Tree-17

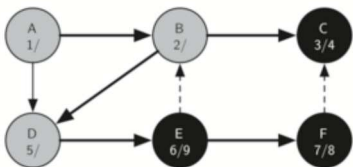


Figure 22: Constructing the Depth First Search Tree-18

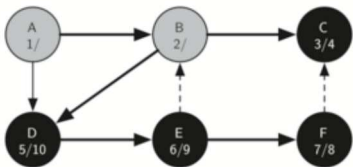


Figure 23: Constructing the Depth First Search Tree-19

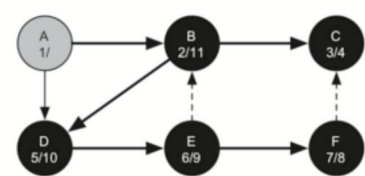


Figure 24: Constructing the Depth First Search Tree-20

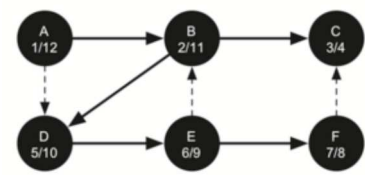


Figure 25: Constructing the Depth First Search Tree-21

The starting and finishing times for each node display a property called the **parenthesis property**. This property means that all the children of a particular node in the depth first tree have a later discovery time and an earlier finish time than their parent. [:ref:`Figure 26 <fig_dfstree>`](#) shows the tree constructed by the depth first search algorithm.

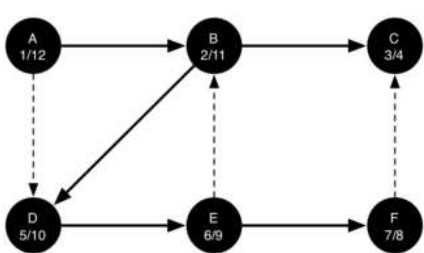




Figure 26: The Resulting Depth First Search Tree

 Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [DepthFirstSearchAnalysis.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

18 lines (12 sloc) 842 Bytes

Depth First Search Analysis

The general running time for depth first search is as follows. The loops in `dfs` both run in $O(V)$, not counting what happens in `dfsvisit`, since they are executed once for each vertex in the graph. In `dfsvisit` the loop is executed once for each edge in the adjacency list of the current vertex. Since `dfsvisit` is only called recursively if the vertex is white, the loop will execute a maximum of once for every edge in the graph or $O(E)$. So, the total time for depth first search is $O(V + E)$.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [TopologicalSorting.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

84 lines (53 sloc) 3.02 KB

Topological Sorting

To demonstrate that computer scientists can turn just about anything into a graph problem, let’s consider the difficult problem of stirring up a batch of pancakes. The recipe is really quite simple: 1 egg, 1 cup of pancake mix, 1 tablespoon oil, and $\frac{3}{4}$ cup of milk. To make pancakes you must heat the griddle, mix all the ingredients together and spoon the mix onto a hot griddle. When the pancakes start to bubble you turn them over and let them cook until they are golden brown on the bottom. Before you eat your pancakes you are going to want to heat up some syrup. [:ref:Figure 27 <fig_pancakes>](#) illustrates this process as a graph.

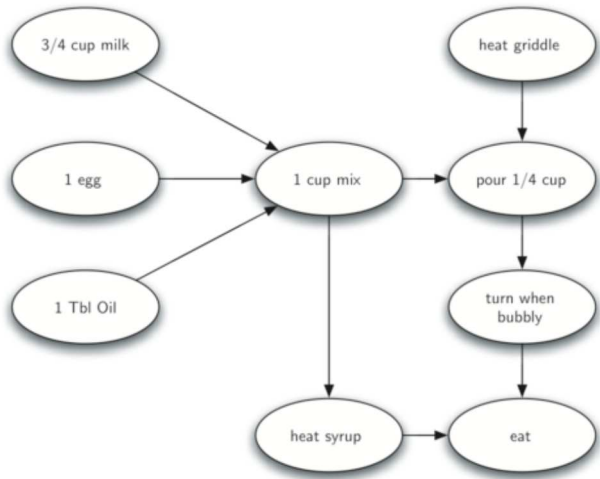


Figure 27: The Steps for Making Pancakes

The difficult thing about making pancakes is knowing what to do first. As you can see from [:ref:Figure 27 <fig_pancakes>](#) you might start by heating the griddle or by adding any of the ingredients to the pancake mix. To help us decide the precise order in which we should do each of the steps required to make our pancakes we turn to a graph algorithm called the **topological sort**.

A topological sort takes a directed acyclic graph and produces a linear ordering of all its vertices such that if the graph G contains an edge (v,w) then the vertex v comes before the vertex w in the ordering. Directed acyclic graphs are used in many applications to indicate the precedence of events. Making pancakes is just one example; other examples include software project schedules, precedence charts for optimizing database queries, and multiplying matrices.

The topological sort is a simple but useful adaptation of a depth first search. The algorithm for the topological sort is as follows:

1. Call $dfs(g)$ for some graph g . The main reason we want to call depth first search is to compute the finish times for

- each of the vertices.
2. Store the vertices in a list in decreasing order of finish time.
3. Return the ordered list as the result of the topological sort.

[:ref:Figure 28 <fig_pancakesDFS>](#) shows the depth first forest constructed by `dfs` on the pancake-making graph shown in [:ref:Figure 26 <fig_pancakes>](#).

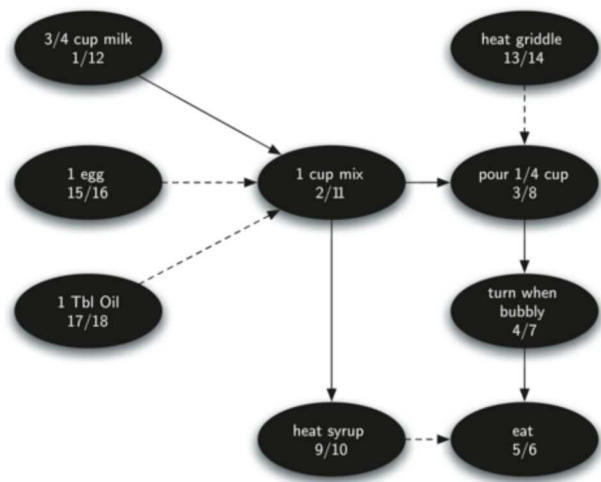


Figure 28: Result of Depth First Search on the Pancake Graph

Finally, [:ref:Figure 29 <fig_pancakesTS>](#) shows the results of applying the topological sort algorithm to our graph. Now all the ambiguity has been removed and we know exactly the order in which to perform the pancake making steps.

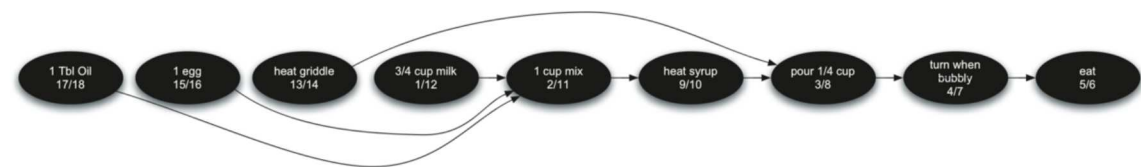
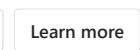


Figure 29: Result of Topological Sort on Directed Acyclic Graph



If you study the graph in [:ref:'Figure 30 <fig_cshome>'](#) you might make some interesting observations. First you might notice that many of the other web sites on the graph are other Luther College web sites. Second, you might notice that there are several links to other colleges in Iowa. Third, you might notice that there are several links to other liberal arts colleges. You might conclude from this that there is some underlying structure to the web that clusters together web sites that are similar on some level.

One graph algorithm that can help find clusters of highly interconnected vertices in a graph is called the strongly connected components algorithm (SCC). We formally define a **strongly connected component**, c , of a graph G , as the largest subset of vertices $c \subseteq V$ such that for every pair of vertices $v, w \in c$ we have a path from v to w and a path from w to v . [:ref:'Figure 27 <fig_scc1>'](#) shows a simple graph with three strongly connected components. The strongly connected components are identified by the different shaded areas.

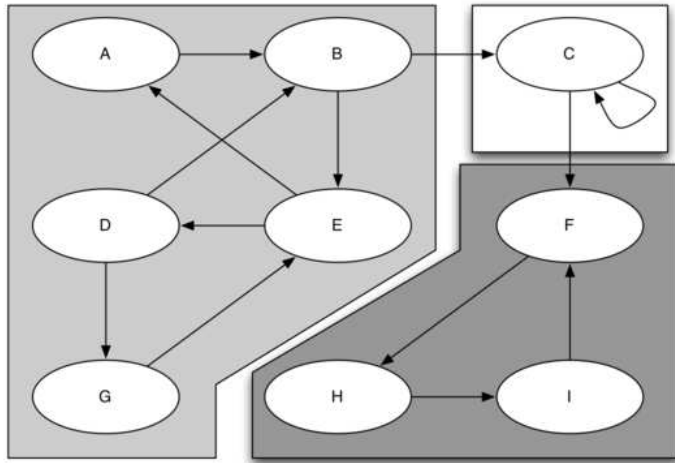


Figure 31: A Directed Graph with Three Strongly Connected Components

Once the strongly connected components have been identified we can show a simplified view of the graph by combining all the vertices in one strongly connected component into a single larger vertex. The simplified version of the graph in [:ref:'Figure 31 <fig_scc1>'](#) is shown in [:ref:'Figure 32 <fig_scc2>'](#).

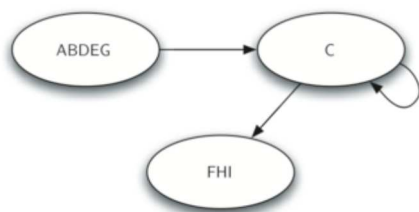


Figure 32: The Reduced Graph

Once again we will see that we can create a very powerful and efficient algorithm by making use of a depth first search. Before we tackle the main SCC algorithm we must look at one other definition. The transposition of a graph G is defined as the graph G^T where all the edges in the graph have been reversed. That is, if there is a directed edge from node A to node B in the original graph then G^T will contain an edge from node B to node A. [:ref:'Figure 33 <fig_tpa>'](#) and [:ref:'Figure 34 <fig_tpb>'](#) show a simple graph and its transposition.

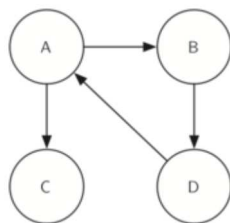


Figure 33: A Graph G

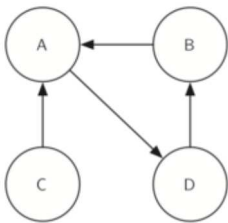


Figure 34: Its Transpose G^T

Look at the figures again. Notice that the graph in [:ref:Figure 33 <fig_tpa>](#) has two strongly connected components. Now look at [:ref:Figure 34 <fig_tpb>](#). Notice that it has the same two strongly connected components.

We can now describe the algorithm to compute the strongly connected components for a graph.

1. Call `dfs` for the graph G to compute the finish times for each vertex.
2. Compute G^T .
3. Call `dfs` for the graph G^T but in the main loop of DFS explore each vertex in decreasing order of finish time.
4. Each tree in the forest computed in step 3 is a strongly connected component. Output the vertex ids for each vertex in each tree in the forest to identify the component.

Let's trace the operation of the steps described above on the example graph in [:ref:Figure 31 <fig_scc1>](#). [:ref:Figure 35 <fig_sccalga>](#) shows the starting and finishing times computed for the original graph by the DFS algorithm. [:ref:Figure 36 <fig_sccalgb>](#) shows the starting and finishing times computed by running DFS on the transposed graph.

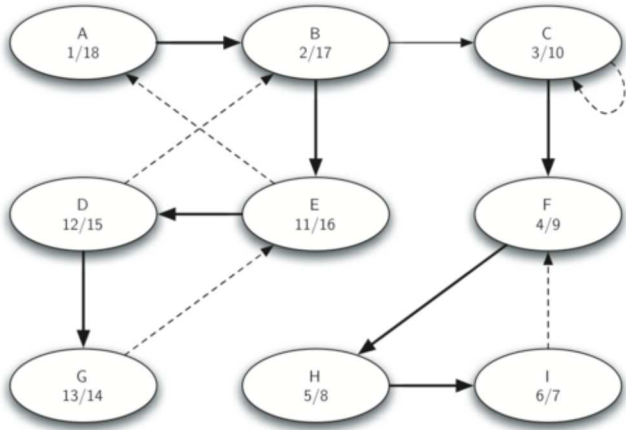


Figure 35: Finishing times for the original graph G

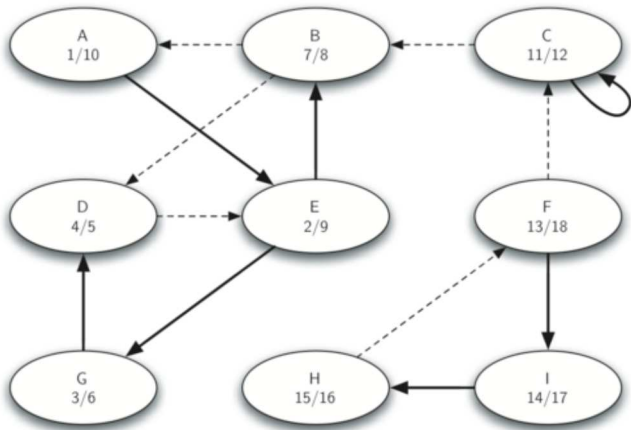


Figure 36: Finishing times for G^T

Finally, [:ref:Figure 37 <fig_sccforest>](#) shows the forest of three trees produced in step 3 of the strongly connected component algorithm. You will notice that we do not provide you with the Python code for the SCC algorithm, we leave writing this program as an exercise.

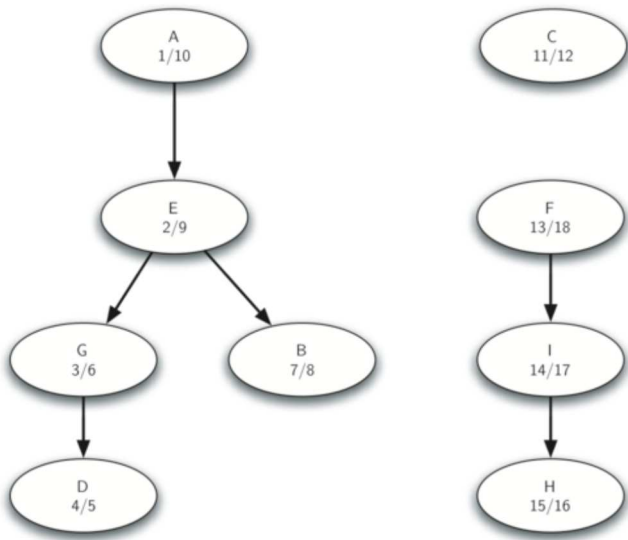


Figure 37: Strongly Connected Components



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: master ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [ShortestPathProblems.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

88 lines (64 sloc) 3.9 KB

Shortest Path Problems

When you surf the web, send an email, or log in to a laboratory computer from another location on campus a lot of work is going on behind the scenes to get the information on your computer transferred to another computer. The in-depth study of how information flows from one computer to another over the Internet is the primary topic for a class in computer networking. However, we will talk about how the Internet works just enough to understand another very important graph algorithm.

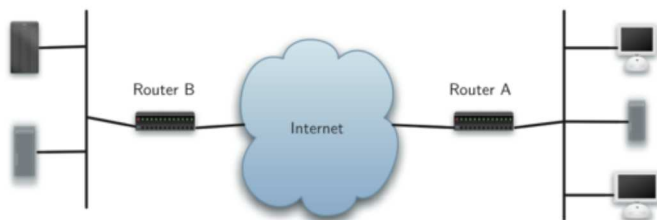


Figure 1: Overview of Connectivity in the Internet

[:ref:Figure 1 <fig_inet>](#) shows you a high-level overview of how communication on the Internet works. When you use your browser to request a web page from a server, the request must travel over your local area network and out onto the Internet through a router. The request travels over the Internet and eventually arrives at a router for the local area network where the server is located. The web page you requested then travels back through the same routers to get to your browser. Inside the cloud labelled "Internet" in [:ref:Figure 1 <fig_inet>](#) are additional routers. The job of all of these routers is to work together to get your information from place to place. You can see there are many routers for yourself if your computer supports the `tracert` command. The text below shows the output of the `tracert` command which illustrates that there are 13 routers between the web server at Luther College and the mail server at the University of Minnesota.

```
1 192.203.196.1
2 hilda.luther.edu (216.159.75.1)
3 ICN-Luther-Ether.icn.state.ia.us (207.165.237.137)
4 ICN-ISP-1.icn.state.ia.us (209.56.255.1)
5 p3-0.hsa1.chi1.bbnplanet.net (4.24.202.13)
6 ae-1-54.bbr2.Chicago1.Level3.net (4.68.101.97)
7 so-3-0-0.mpls2.Minneapolis1.Level3.net (64.159.4.214)
8 ge-3-0.hsa2.Minneapolis1.Level3.net (4.68.112.18)
9 p1-0.minnesota.bbnplanet.net (4.24.226.74)
10 TelecomB-BR-01-V4002.ggnnet.umn.edu (192.42.152.37)
11 TelecomB-BN-01-Vlan-3000.ggnnet.umn.edu (128.101.58.1)
12 TelecomB-CN-01-Vlan-710.ggnnet.umn.edu (128.101.80.158)
13 baldrick.cs.umn.edu (128.101.80.129) (N!) 88.631 ms (N!)
```

Routers from One Host to the Next over the Internet

Each router on the Internet is connected to one or more other routers. So if you run the `traceroute` command at different times of the day, you are likely to see that your information flows through different routers at different times. This is because there is a cost associated with each connection between a pair of routers that depends on the volume of traffic, the time of day, and many other factors. By this time it will not surprise you to learn that we can represent the network of routers as a graph with weighted edges.

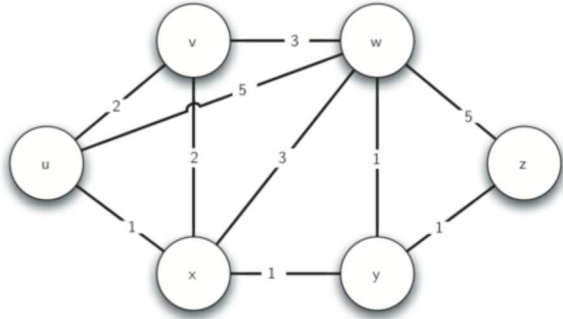


Figure 2: Connections and Weights between Routers in the Internet

[:ref:Figure 2 <fig_network>](#) shows a small example of a weighted graph that represents the interconnection of routers in the Internet. The problem that we want to solve is to find the path with the smallest total weight along which to route any given message. This problem should sound familiar because it is similar to the problem we solved using a breadth first search, except that here we are concerned with the total weight of the path rather than the number of hops in the path. It should be noted that if all the weights are equal, the problem is the same.



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: [master](#) ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [DijkstrasAlgorithm.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

162 lines (121 sloc) 6.99 KB

Dijkstra's Algorithm

The algorithm we are going to use to determine the shortest path is called "Dijkstra's algorithm." Dijkstra's algorithm is an iterative algorithm that provides us with the shortest path from one particular starting node to all other nodes in the graph. Again this is similar to the results of a breadth first search.

To keep track of the total cost from the start node to each destination we will make use of the `dist` instance variable in the `Vertex` class. The `dist` instance variable will contain the current total weight of the smallest weight path from the start to the vertex in question. The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by a priority queue. The value that is used to determine the order of the objects in the priority queue is `dist`. When a vertex is first created `dist` is set to a very large number. Theoretically you would set `dist` to infinity, but in practice we just set it to a number that is larger than any real distance we would have in the problem we are trying to solve.

The code for Dijkstra's algorithm is shown in [:ref:'Listing 1 <lst_shortpath>'](#). When the algorithm finishes the distances are set correctly as are the predecessor links for each vertex in the graph.

Listing 1

```
from pythonds.graphs import PriorityQueue, Graph, Vertex
def dijkstra(aGraph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance( newDist )
                nextVert.setPred(currentVert)
        pq.decreaseKey(nextVert,newDist)
```

Dijkstra's algorithm uses a priority queue. You may recall that a priority queue is based on the heap that we implemented in the `Tree` Chapter. There are a couple of differences between that simple implementation and the implementation we use for Dijkstra's algorithm. First, the `PriorityQueue` class stores tuples of key, value pairs. This is important for Dijkstra's algorithm as the key in the priority queue must match the key of the vertex in the graph. Secondly the value is used for deciding the priority, and thus the position of the key in the priority queue. In this implementation we use the distance to the vertex as the priority because as we will see when we are exploring the next vertex, we always want to explore the vertex that has the smallest distance. The second difference is the addition of the `decreaseKey` method. As you can see, this method is used when the distance to a vertex that is already in the queue is reduced, and thus moves that vertex toward the front of the queue.

Let's walk through an application of Dijkstra's algorithm one vertex at a time using the following sequence of figures as our guide. We begin with the vertex u . The three vertices adjacent to u are v , w , and x . Since the initial distances to v , w , and x are all initialized to `sys.maxint`, the new costs to get to them through the start node are all their direct costs. So we update the costs to each of these three nodes. We also set the predecessor for each node to u and we add each node to the priority queue. We use the distance as the key for the priority queue. The state of the algorithm is shown in :ref:`Figure 3 <fig_dija>`.

In the next iteration of the `while` loop we examine the vertices that are adjacent to x . The vertex x is next because it has the lowest overall cost and therefore bubbled its way to the beginning of the priority queue. At x we look at its neighbors u , v , w and y . For each neighboring vertex we check to see if the distance to that vertex through x is smaller than the previously known distance. Obviously this is the case for y since its distance was `sys.maxint`. It is not the case for u or v since their distances are 0 and 2 respectively. However, we now learn that the distance to w is smaller if we go through x than from u directly to w . Since that is the case we update w with a new distance and change the predecessor for w from u to x . See :ref:`Figure 4 <fig_dijb>` for the state of all the vertices.

The next step is to look at the vertices neighboring v (see :ref:`Figure 5 <fig_dijc>`). This step results in no changes to the graph, so we move on to node y . At node y (see :ref:`Figure 6 <fig_dijd>`) we discover that it is cheaper to get to both w and z , so we adjust the distances and predecessor links accordingly. Finally we check nodes w and z (see :ref:`Figure 6 <fig_dije>` and see :ref:`Figure 8 <fig_dijf>`). However, no additional changes are found and so the priority queue is empty and Dijkstra's algorithm exits.

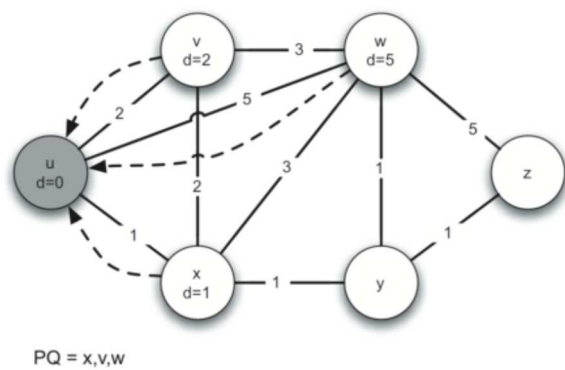


Figure 3: Tracing Dijkstra's Algorithm

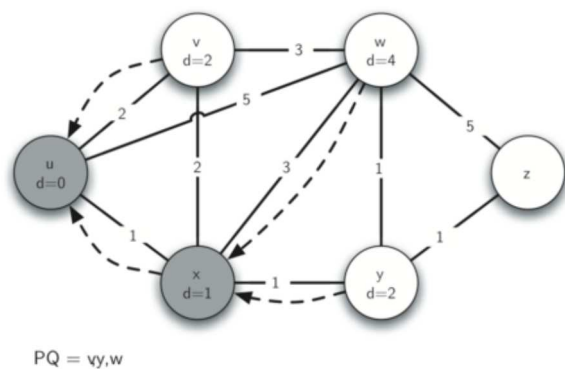


Figure 4: Tracing Dijkstra's Algorithm

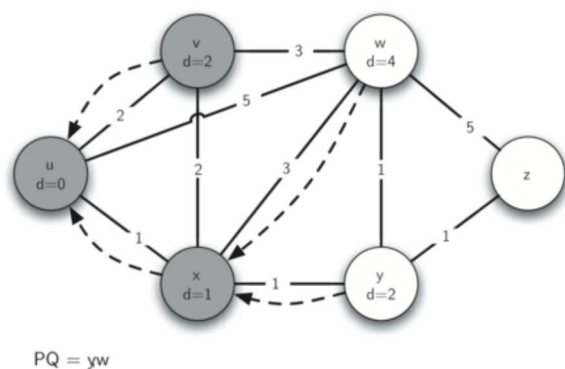


Figure 5: Tracing Dijkstra's Algorithm

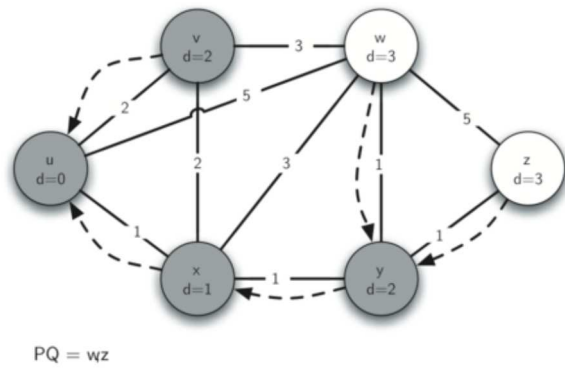


Figure 6: Tracing Dijkstra's Algorithm

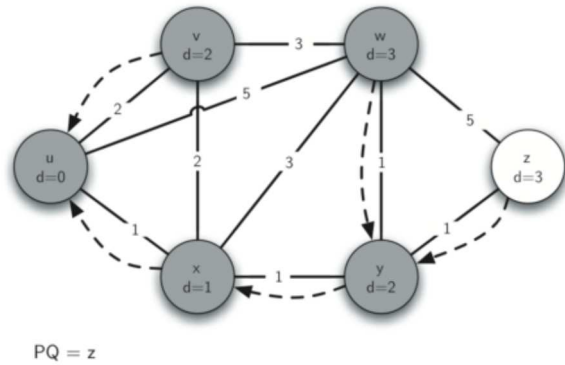


Figure 7: Tracing Dijkstra's Algorithm

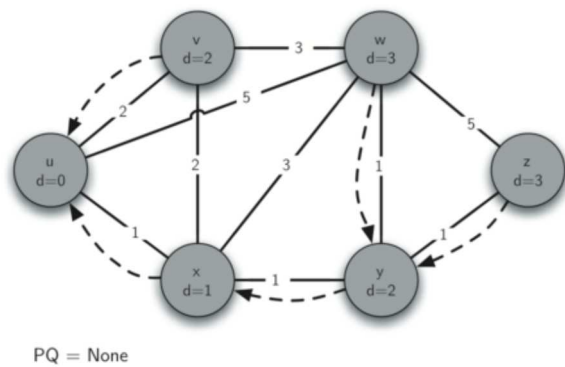



Figure 8: Tracing Dijkstra's Algorithm

It is important to note that Dijkstra's algorithm works only when the weights are all positive. You should convince yourself that if you introduced a negative weight on one of the edges to the graph that the algorithm would never exit.

We will note that to route messages through the Internet, other algorithms are used for finding the shortest path. One of the problems with using Dijkstra's algorithm on the Internet is that you must have a complete representation of the graph in order for the algorithm to run. The implication of this is that every router has a complete map of all the routers in the Internet. In practice this is not the case and other variations of the algorithm allow each router to discover the graph as they go. One such algorithm that you may want to read about is called the "distance vector" routing algorithm.

 Please note that GitHub no longer supports old versions of Firefox. We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [AnalysisofDijkstrasAlgorithm.rst](#)

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

21 lines (15 sloc) 1.04 KB

Analysis of Dijkstra’s Algorithm

Finally, let us look at the running time of Dijkstra’s algorithm. We first note that building the priority queue takes $O(V)$ time since we initially add every vertex in the graph to the priority queue. Once the queue is constructed the `while` loop is executed once for every vertex since vertices are all added at the beginning and only removed after that. Within that loop each call to `delMin`, takes $O(\log V)$ time. Taken together that part of the loop and the calls to `delMin` take $O(V \log(V))$. The `for` loop is executed once for each edge in the graph, and within the `for` loop the call to `decreaseKey` takes time $O(\log(V))$. So the combined running time is $O((V+E) \log(V))$.

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

pythonds / _sources / Graphs / PrimsSpanningTreeAlgorithm.rst

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

202 lines (151 sloc) 8.03 KB

Prim’s Spanning Tree Algorithm

For our last graph algorithm let’s consider a problem that online game designers and Internet radio providers face. The problem is that they want to efficiently transfer a piece of information to anyone and everyone who may be listening. This is important in gaming so that all the players know the very latest position of every other player. This is important for Internet radio so that all the listeners that are tuned in are getting all the data they need to reconstruct the song they are listening to. [:ref:Figure 9 <fig_bcast1>](#) illustrates the broadcast problem.

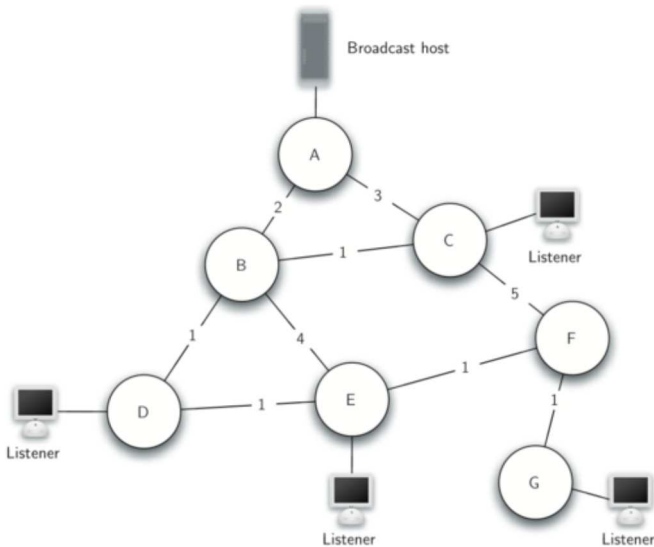


Figure 9: The Broadcast Problem

There are some brute force solutions to this problem, so let’s look at them first to help understand the broadcast problem better. This will also help you appreciate the solution that we will propose when we are done. To begin, the broadcast host has some information that the listeners all need to receive. The simplest solution is for the broadcasting host to keep a list of all of the listeners and send individual messages to each. In [:ref:Figure 9 <fig_bcast1>](#) we show a small network with a broadcaster and some listeners. Using this first approach, four copies of every message would be sent. Assuming that the least cost path is used, let’s see how many times each router would handle the same message.

All messages from the broadcaster go through router A, so A sees all four copies of every message. Router C sees only one copy of each message for its listener. However, routers B and D would see three copies of every message since routers B and D are on the cheapest path for listeners 1, 2, and 3. When you consider that the broadcast host must send hundreds of messages each second for a radio broadcast, that is a lot of extra traffic.

A brute force solution is for the broadcast host to send a single copy of the broadcast message and let the routers sort things out. In this case, the easiest solution is a strategy called **uncontrolled flooding**. The flooding strategy works as follows. Each message starts with a time to live (`ttl`) value set to some number greater than or equal to the number of edges between the broadcast host and its most distant listener. Each router gets a copy of the message and passes the message on to *all* of its neighboring routers. When the message is passed on the `ttl` is decreased. Each router continues to send copies of the message to all its neighbors until the `ttl` value reaches 0. It is easy to convince yourself that uncontrolled flooding generates many more unnecessary messages than our first strategy.

The solution to this problem lies in the construction of a minimum weight **spanning tree**. Formally we define the minimum spanning tree T for a graph $G = (V, E)$ as follows. T is an acyclic subset of E that connects all the vertices in V . The sum of the weights of the edges in T is minimized.

[:ref:'Figure 10 <fig_mst1>'](#) shows a simplified version of the broadcast graph and highlights the edges that form a minimum spanning tree for the graph. Now to solve our broadcast problem, the broadcast host simply sends a single copy of the broadcast message into the network. Each router forwards the message to any neighbor that is part of the spanning tree, excluding the neighbor that just sent it the message. In this example A forwards the message to B. B forwards the message to D and C. D forwards the message to E, which forwards it to F, which forwards it to G. No router sees more than one copy of any message, and all the listeners that are interested see a copy of the message.

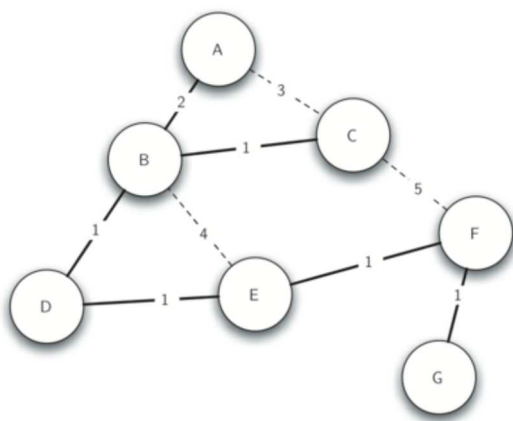


Figure 10: Minimum Spanning Tree for the Broadcast Graph

The algorithm we will use to solve this problem is called Prim's algorithm. Prim's algorithm belongs to a family of algorithms called the "greedy algorithms" because at each step we will choose the cheapest next step. In this case the cheapest next step is to follow the edge with the lowest weight. Our last step is to develop Prim's algorithm.

The basic idea in constructing a spanning tree is as follows:

```
While T is not yet a spanning tree
    Find an edge that is safe to add to the tree
    Add the new edge to T
```

The trick is in the step that directs us to "find an edge that is safe." We define a safe edge as any edge that connects a vertex that is in the spanning tree to a vertex that is not in the spanning tree. This ensures that the tree will always remain a tree and therefore have no cycles.

The Python code to implement Prim's algorithm is shown in [:ref:'Listing 2 <lst_prims>'](#). Prim's algorithm is similar to Dijkstra's algorithm in that they both use a priority queue to select the next vertex to add to the growing graph.

Listing 2

```
from pythonds.graphs import PriorityQueue, Graph, Vertex

def prim(G, start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
```

```

newCost = currentVert.getWeight(nextVert)
if nextVert in pq and newCost < nextVert.getDistance():
    nextVert.setPred(currentVert)
    nextVert.setDistance(newCost)
    pq.decreaseKey(nextVert, newCost)

```

The following sequence of figures (:ref:`Figure 11 <fig_mst1>` through :ref:`Figure 17 <fig_mst1>`) shows the algorithm in operation on our sample tree. We begin with the starting vertex as A. The distances to all the other vertices are initialized to infinity. Looking at the neighbors of A we can update distances to two of the additional vertices B and C because the distances to B and C through A are less than infinity. This moves B and C to the front of the priority queue. Update the predecessor links for B and C by setting them to point to A. It is important to note that we have not formally added B or C to the spanning tree yet. A node is not considered to be part of the spanning tree until it is removed from the priority queue.

Since B has the smallest distance we look at B next. Examining B's neighbors we see that D and E can be updated. Both D and E get new distance values and their predecessor links are updated. Moving on to the next node in the priority queue we find C. The only node C is adjacent to that is still in the priority queue is F, thus we can update the distance to F and adjust F's position in the priority queue.

Now we examine the vertices adjacent to node D. We find that we can update E and reduce the distance to E from 6 to 4. When we do this we change the predecessor link on E to point back to D, thus preparing it to be grafted into the spanning tree but in a different location. The rest of the algorithm proceeds as you would expect, adding each new node to the tree.

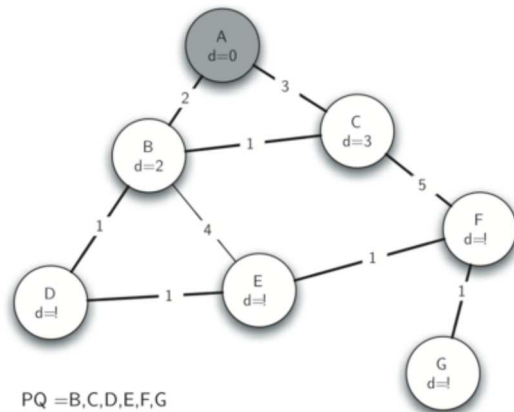


Figure 11: Tracing Prim's Algorithm

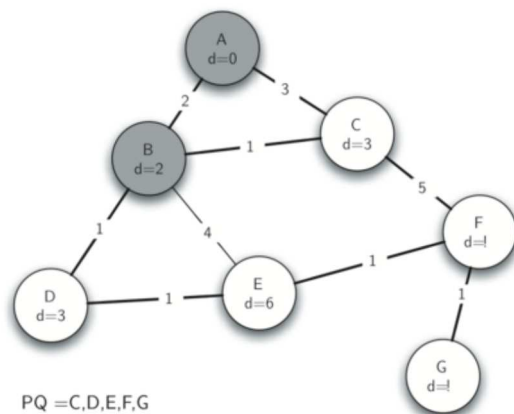


Figure 12: Tracing Prim's Algorithm

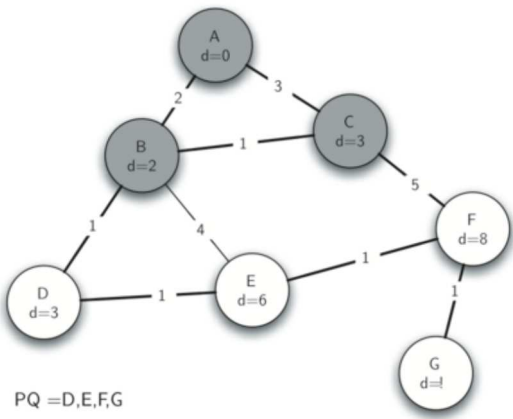


Figure 13: Tracing Prim's Algorithm

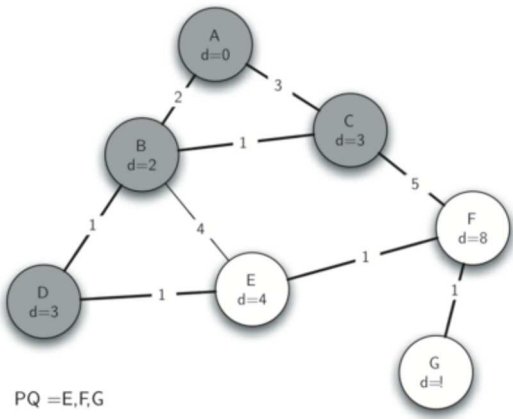


Figure 14: Tracing Prim's Algorithm

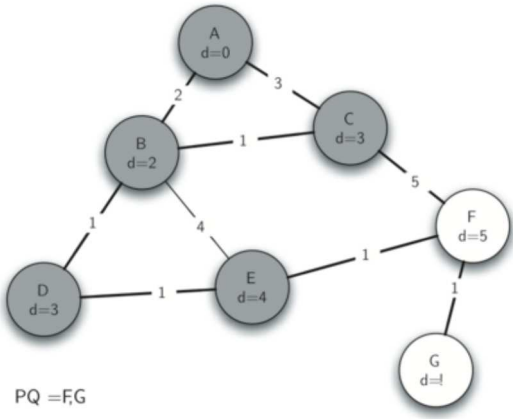


Figure 15: Tracing Prim's Algorithm

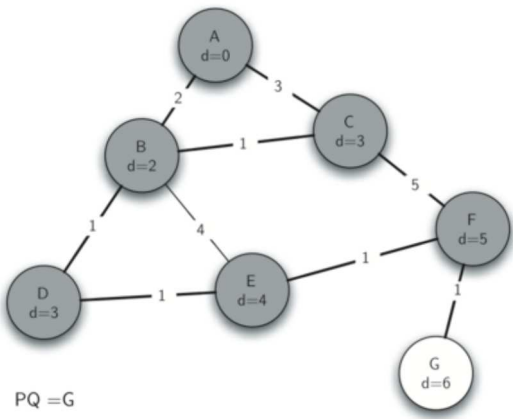


Figure 16: Tracing Prim's Algorithm

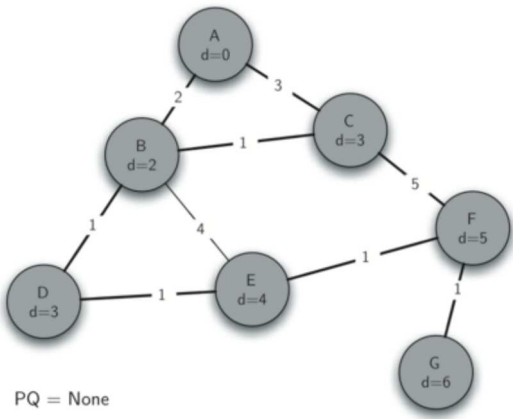



Figure 17: Tracing Prim's Algorithm

 Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [Summary.rst](#)

Find file

Copy path

Fetching contributors...


Cannot retrieve contributors at this time.

26 lines (15 sloc) 941 Bytes

Summary


In this chapter we have looked at the graph abstract data type, and some implementations of a graph. A graph enables us to solve many problems provided we can transform the original problem into something that can be represented by a graph. In particular, we have seen that graphs are useful to solve problems in the following general areas.

- Breadth first search for finding the unweighted shortest path.
- Dijkstra’s algorithm for weighted shortest path.
- Depth first search for graph exploration.
- Strongly connected components for simplifying a graph.
- Topological sort for ordering tasks.
- Minimum weight spanning trees for broadcasting messages.

 Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

 RunestoneInteractive / [pythonds](#)

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

[pythonds](#) / [_sources](#) / [Graphs](#) / [KeyTerms.rst](#)

Find file

Copy path


Fetching contributors...

Cannot retrieve contributors at this time.

20 lines (14 sloc) 1.16 KB

Key Terms

acyclic graph	adjacency list	adjacency matrix
adjacent	breadth first search (BFS)	cycle
cyclic graph	DAG	depth first forest
depth first search (DFS)	digraph	directed acyclic graph (DAG)
directed graph	edge cost	edge
parenthesis property	path	shortest path
spanning tree	strongly connected components (SCC)	topological sort & uncontrolled flooding
vertex	weight	

 Please note that GitHub no longer supports old versions of Firefox. We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

Ignore

Learn more

RunestoneInteractive / pythonds

Dismiss

Join GitHub today

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

Sign up

Branch: master ▾

pythonds / _sources / Graphs / DiscussionQuestions.rst

Find file

Copy path

Fetching contributors...

Cannot retrieve contributors at this time.

67 lines (47 sloc) 2.16 KB

Discussion Questions

1. Draw the graph corresponding to the following adjacency matrix.

	A	B	C	D	E	F
A		7	5			1
B	2			7	3	
C		2				8
D	1				2	4
E	6			5		
F		1			8	

1. Draw the graph corresponding to the following list of edges.

from	to	cost
1	2	10
1	3	15
1	6	5
2	3	7
3	4	7
3	6	10

from	to	cost
4	5	7
6	4	5
5	6	13

2. Ignoring the weights, perform a breadth first search on the graph from the previous question.
3. What is the Big-O running time of the `buildGraph` function?
4. Derive the Big-O running time for the topological sort algorithm.
5. Derive the Big-O running time for the strongly connected components algorithm.
6. Show each step in applying Dijkstra's algorithm to the graph shown above.
7. Using Prim's algorithm, find the minimum weight spanning tree for the graph shown above.
8. Draw a dependency graph illustrating the steps needed to send an email. Perform a topological sort on your graph.
9. Derive an expression for the base of the exponent used in expressing the running time of the knights tour.
10. Explain why the general DFS algorithm is not suitable for solving the knights tour problem.
11. What is the Big-O running time for Prim's minimum spanning tree algorithm?



Please note that GitHub no longer supports old versions of Firefox.
We recommend upgrading to the latest [Safari](#), [Google Chrome](#), or [Firefox](#).

[Ignore](#)[Learn more](#)[RunestoneInteractive](#) / [pythonds](#)

Join GitHub today

[Dismiss](#)

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)Branch: [master](#) ▾ [pythonds](#) / [_sources](#) / [Graphs](#) / [ProgrammingExercises.rst](#)[Find file](#)[Copy path](#)

Fetching contributors...

Cannot retrieve contributors at this time.

39 lines (27 sloc) 1.65 KB

Programming Exercises

1. Modify the depth first search function to produce a topological sort.
2. Modify the depth first search to produce strongly connected components.
3. Write the `transpose` method for the `Graph` class.
4. Using breadth first search write an algorithm that can determine the shortest path from each vertex to every other vertex. This is called the all pairs shortest path problem.
5. Using breadth first search revise the maze program from the recursion chapter to find the shortest path out of a maze.
6. Write a program to solve the following problem: You have two jugs, a 4-gallon and a 3-gallon. Neither of the jugs has markings on them. There is a pump that can be used to fill the jugs with water. How can you get exactly two gallons of water in the 4 gallon jug?
7. Generalize the problem above so that the parameters to your solution include the sizes of each jug and the final amount of water to be left in the larger jug.
8. Write a program that solves the following problem: Three missionaries and three cannibals come to a river and find a boat that holds two people. Everyone must get across the river to continue on the journey. However, if the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. Find a series of crossings that will get everyone safely to the other side of the river.